PRIME Technical Update

Subject: Subroutines, Rev. 18

Number: 78

Revision: 0

Date: December 1980

Applicable Hardware: All CPUs

Applicable Software: PRIMOS

Documentation Impact: Subroutines Reference Guide (PDR3621)

Abstract: The new file naming convention and suffixes are discussed. Subroutines supporting this convention, APSFX$ and SRSFX$, are included.

The subroutines GV$SET and GV$GET supporting global variables are given.

There is now a V-mode versions of the in-memory sort library -- VMSORT.

Named semaphores have been added to PRIMOS. This topic is covered in detail along with supporting subroutines.

New conditions have been added to the condition mechanism.

There are minor changes, additional keys, and typographic corrections to a number of subroutines.

PRIME Computer, Inc. 145 Pennsylvania Avenue, Framingham, Mass. 01701
(617) 879-2960

PTU 78

REV. 18 SUBROUTINES

ADD TO PAGE 1-2

THE FILE NAMING CONVENTION

Beginning at Rev. 18, a new file naming convention is being adopted by PRIMOS. In this convention, file names may be seprated into one or more components by the separator '.'. For example, the name A.BCD.EFGH has three components: A, BCD and EFGH.

The last component of a name may be a standard suffix. Standard suffixes identify the primary purpose of the file. The remainder of the name is called the base name, and is the file's principal identifier. For example, F77.PL1 has the standard suffix PL1, which indicates that the file is a PL1 source file.

If two or more files in a directory have the same base name but different suffixes, it can be assumed they are related files of the same type: e.g. F77.PL1, F77.LIST, F77.BIN are the source, listing and binary files for the PL1 program F77.

If two or more files in a directory have the same suffix but different base names, they are unrelated files of the same type: e.g. F77.PL1 and BAR.PL1 are both PL1 source files.

Certain Prime software has been changed at Rev. 18 to support this convention, notably the compilers, loaders and RESUME command. See the updates to the language reference guides and the update to the LOAD and SEG Guide for more details.

****

ADD TO PAGE 3-11

The R-mode FTNLIB has been duplicated into two files FTNLIB and SVCLIB

****

ADD TO PAGE 3-11

Add the following modification to Segment Directory Formatting.

Segment directories are limited to 64K words and, therefore 32K entries.

****

ADD TO PAGE 4-3

Add the following parameter to TRSC$$ subroutine.

CALL TSRC$$ (action+newfil, treename, funit, chrpos, type, code)

K$GETU   Open treename on an unused file unit selected by PRIMOS

The unit number is returned in funit.

****

ADD TO PAGE 4-3

APSFX$

The direct call APSFX$ appends a specified suffix to a pathname. The pathname is checked for the prior existence of the suffix to avoid overwriting on existing file.

DCL APSFX$ ENTRY (char(128) var, char(128) var, char(32) var, fixed bin);

CALL APSFX$ (in_pathname, out_pathname, suffix, status);

in pathname   Pathname to use to process suffix checking. (Input)

out pathname   Pathname returned to caller with desired suffix appended. (Output)

suffix   This is the suffix desired on the pathname. It should include the period, and be capital letters, e.g., ".F77". (Input)

status   -1 - suffix already present, pathname remained untouched. Ø - suffix appended ok. e$nmlg - pathname+suffix > 128 chars or filename+suffix is longer than 32 chars. (Output)

****

ADD TO PAGE 4-7

The command output unit depends on the FILUNT directive in the CONFIG file at cold start.

****

ADD TO PAGE 4-27

In subroutine SRCH$$ the sub-key, K$GETU, returns the PRIMOS file unit (not FORTRAN unit) in funit.

****

ADD TO PAGE 4-27

SRSFX$

The direct call SRSFX$ searches for a file according to the file naming standards. In addition to the normal search arguments, key, pathname, unit, and type, the caller supplies a list of suffixes.

The suffix list defines both the suffixes to scan for and the search order. If the suffix already exists at the end of the filename then a tree search is performed with the pathname as is.

If none of the desired suffixes are found, a tree search is performed in the following manner: attach to the appropriate directory, each suffix in the list is appended to the filename a search is done. In this way the suffix list defines the search order. The routine returns if a "filename.suffix" is found or the suffix list is exhausted.

If a file is found, the index (in the suffix list) of the last suffix in the filename is returned; if no file is found, or if none of the suffixes in the list is on the found filename, an index of zero is returned.

Restrictions and notes:

1. The null string is not allowed as an element of the suffix list. The null suffix is assumed if no desired suffix is found. In this case the suffix index is set to zero and a processor may then chose to use old-style conventions, B_, L_ etc., for its output files.

2. If the suffix list contains ".F77", a pathname such as "pathname>.F77" will be treated as a valid suffix found, i.e., ".F77". The filename returned will be '', the null string.

3. If the filename + suffix > 32 chars or pathname + suffix > 128 chars, a search with suffix will not be done and the next suffix is attempted. E.g. a filename = 32 chars will simply be searched as is.

4. The suffixes in the suffix list, provided by the caller, must contain the period and be all capital letters, e.g. ".F77".

```
DCL SRSFX$ ENTRY (fixed bin, char(*) var, fixed bin, fixed bin,
                  fixed bin, (*) char(32) var, char(32) var,
                  fixed bin, fixed bin)

[ returns(fixed bin(31)); ] (only used with the function call)

CALL SRSFX$ (key, pathname, unit, type, n_suffixes, suffix_list,
            basename, suffix_used, status);

CHRPOS = SRSFX$ (key, pathname, unit, type, n_suffixes,
                suffix_list, basename, suffix_used, status);
```

key — Key(s) to use for the search. (Input)

pathname — Pathname to use for search (remains unchanged). (Input)

unit — File unit opened, with K$GETU. (Output)
File unit to use for SRCH$$ action, i.e. without K$GETU. (Input)

type — File type opened. (Output)

n_suffixes — Number of suffixes in suffix_list. A value of zero would indicate not to use the file naming standards with suffixes for the search. (Input)

suffix_list — List of desired suffixes to use. Each suffix should include the period and be capital letters, i.e., suffix_list(i) = ".F77". (Input)

basename — This is the base filename, i.e., without a suffix according to the suffix_list. This is useful to callers that want to append a different suffix to the base filename. For example, FTN PROG.TEST.FTN would produce output files with "PROG.TEST" as the basename used, i.e., "PROG.TEST.BIN". (Output)

suffix_used — The is the index, in the suffix list given, of the suffix used for the search. As mentioned, a value of zero denotes that the null suffix was used. (Output)

status — Status from the search operation. (Output)

chrpos — When SRSFX$ is used as a function call this is returned. The first word points one past the pathname component that caused the error. The second word is the pathname length. (Output)

****

ADD TO PAGE 5-1

GCHAR

Gets a character from an array.

CHAR = GCHAR (LOC (array), ptr)

array   Array of characters

ptr   Pointer to the location of character in array

The pointer is origined at zero and is incremented by 1 after the operation is complete. (INT*2)

SCHAR

Stores a character into an array location.

CALL SCHAR (LOC (array), prt, char)

array   Array of characters

ptr   Pointer to the location of character in array

char   Character to be stored

The pointer is origined at zero and is incremented by 1 after the operation is complete. (INT*2)

The right half of the character word is stored, so the form of char should be ' A', for example.

I$AA12

Performs the same function as I$AA01, but also allows the input to be f
rom a
cominput file.

CALL I$AA12 (0, buff, #words)

buff   Buffer location

#words   Number of words to input

GLOBAL VARIABLES

Two routines are available for the accessing and setting of global

variables from inside a user program. GV$SET sets the value of a global variable, and GV$GET retrieves the value.

The GV$SET and GV$GET routines use PL/I data types and declaration statements.

The PRIMOS command DEFINE_GVAR must be used to define the global variable file before either of these two procedures is called.

****

ADD TO PAGE 5-1

LOGOUT NOTIFICATION

Logout Notification provides the creator of a Phantom process information about the Phantom's activites. This information is compiled at Phantom logout time and sent to the Spawner. This is known as notification.

Normally, the information will be displayed upon the creator's terminal. The information contains the Phantom's user number, the time of day of logout, the elapsed time, cpu time, and I/O time spent by the Phantom, and an error code indicating normal or abnormal logout. Normal logout occurs when a phantom completes with a LOGOUT command. All other logout will generate abnormal logout.

Logout information will be compiled at this time and sent to the proper Spawner. If a user is logged into more than one terminal, the information will only be sent to the terminal from which the Phantom was created. If the Spawner of the Phantom has logged out while the Phantom was running, the information will not be sent. This means that once a user has logged out, the Phantom will not notify the user of logout even if the user was to log back in.

Sometimes it may become necessary for a user to record the Phantom logout information via a program. The Logout Notification system provides two subroutines that allow for this event. The first subroutine, LO$CN, allows a user to turn off and/or on Logout Notification. The second subroutine, LO$R, allows a user to fetch Phantom logout information instead of having the information written to their terminal.

When LO$CN is requested to turn off Logout Notification, Phantom logout information is automatically queued for future access. This allows any user to turn off notification without having to worry about either the condition of their terminal screen or the loss of the status of their Phantoms.

When LO$CN is requested to turn on Logout Notification, Logout Notification is enabled and any enqueued logout information is written on the user's terminal.

As was mentioned above, a user may fetch Phantom logout information by invoking LO$R. Normally, Logout Notification is enabled and invoking LO$R will gain the user nothing. Therefore, when using LO$R, Logout Notification should be turned off by invoking LO$CN.

When Logout Notification occurs a system default "on-condition" handler named PH_LO$ is invoked to write the information upon the creator's terminal. This "on-unit" is also invoked when LO$CN is requested to turn on Logout Notification. Therefore, if a user does not ever wish to see logout information written upon their terminal, they should create their own "on-unit" and name it PH_LO$. This user defined PH_LO$ may wish to call LO$R to fetch Phantom logout information. The default PH_LO$ does this.

****

ADD TO PAGE 5-1

LIMIT$

Allows the setting of various timers within PRIMOS each generating a signal if expired. The timer values may also be read.

calling sequence

```
CALL LIMIT$ (KEY,LIMIT,RES,CODE)
```

Parameters are INTEGER*2 except LIMIT which is INTEGER*4

KEY — this key is split into two 8 bit functions. The right half is as follows.
- 1 = read limit
- 2 = set limit

The left half is as follows:
- 1 = cpu limit in seconds
- 2 = login limit in minutes
- 5 = CPU watchdog in seconds
- 6 = real time watchdog in minutes

LIMIT — is the time to be set in minutes or seconds

RES — reserved - must be zero

CODE — is a returned standard error code

****

ADD TO PAGE 5-8

GV$SET

GV$SET sets the value of a global variable. Its calling sequence is:

```
DCL GV$SET ENTRY (CHAR(*) VAR, CHAR(*) VAR, FIXED BIN)

CALL GV$SET (var-name, var-value, code)
```

var-name — is the name of the global variable to be set. This name must follow the rules for CPL global variable names. All letters must be upper case.

var-value — is the new value of the variable var-name.

code — is a return error code. E$BFTS is returned if the specified value is too big. E$UNOP is returned if the global variable area is bad or uninitialized. E$ROOM is returned if an attempt to acquire more storage by the variable management routines fails.

GV$GET

GV$GET retrieves the value of a global variable. Its calling sequence is:

```
DCL GV$GET ENTRY (CHAR(*) VAR, CHAR(*) VAR, FIXED BIN, FIXED BIN)

CALL GV$GET (var-name, var-value, value-size, code)
```

var-name — is the name of the global variable whose value is to be retrieved. The name must follow the rules for CPL global variable names and must be in upper case.

var-value — is returned value of variable var-name.

value-size — is the length of the user's buffer var-value in characters.

code — is a return error code. E$BFTS is returned if the user buffer var-value is too small to hold the current value of the variable. E$UNOP is returned if the global variable storage is uninitialized or in bad format. E$FNTF is returned if the variable is not found.

LO$R

LO$R fetches or transfer logout information from storage to a designated area. It will do this unless it finds no information to transfer. The area to transfer the information to is designated by an arguement to LO$R known as MSGPTR. The size of the area pointed to by MSGPTR is designated by a second arguement known as MSGLEN. The area should be of at least six words in length. If it is shorter than this, LO$R will only fetch as much information as MSGLEN.

LO$R also passes back to its caller an indication that their have been more Phantom logouts with their respective information stored in a queue. This indication is contained within the arguement named MORE.

An error code that indicates the above situation or no error is passed to the user via an arguement named CODE.

```
DCL LO$R ENTRY (pointer,fixed bin(15),bit(1),fixed bin(15));

CALL LO$R (msgptr,msglen,more,code);
```

msgptr	area of memory in which to store message
msglen	length of area in which to store message
more	Ø indicates no more messages left on queue 1 indicate more messages left on queue
code	error code E$NDAT-no data found in queue E$BFTS-Some information lost during

transfer (msglen less than actual

message length)

Information Format

Word Number	Information
1	Phantom's user number (fixed bin(15))
2	Time of day of logout (fixed bin(15))
3	Connect time in minutes (fixed bin(15))
4	CPU time in seconds (fixed bin(15))
5	I/O time in seconds (fixed bin(15))
6	Logout condition code (fixed bin(15)) Ø-Normal Logout 1-Abnormal Logout

LO$CN

LO$CN is used to either turn off or turn on Logout Notification. When notification is turned off, Phantom logout information is queued on a FIFO basis. When notification is turned on, queuing will not be performed, and the Logout Notification default "on-condition", PH_LO$, will be raised if there is any Logout Notification data to be received.

DCL LO$CN ENTRY (fixed bin(15));

DCL LO$CN (key);

key software interrupt status key

Ø-notify off
1-notify on

****

CHANGE TO PAGE 7-5

Change the Argument type for RND subroutine from INTEGER to REAL.

****

ADD TO PAGE 11-23

The value argument for CNVA$A is returned as Integer*4.

value Returned converted binary value (INTEGER*4).

ADD TO PAGE 11-26

Whenever the APPLIB key, A$GETU, is specified, the PRIMOS file unit is returned in the user's argument unit.

****

ADD TO PAGE 11-29

OPEN$A untkey should read "choose a PRIMOS file unit"

****

ADD TO PAGE 11-3Ø

OPNP$A untkey should read "choose a PRIMOS file unit"

OPNP$A unit should read "PRIMOS file unit"

OPNV$A untkey should read "choose a PRIMOS file unit"

****

ADD TO PAGE 11-31

OPNP$A unit should read "PRIMOS file unit"

****

ADD TO SECTION 12 - SORT LIBRARIES

SORT and VSRTLI will now handle up to 64 keys as required by SPSS (Statistical Package for the Social Sciences).

VSRTLI

In V-mode up to 64 key fields may be specified. The parameter numkey description also changes in the following subroutines: SUBSRT ASCSRT SRTF$S MRG1$S SETU$S

Key Definitions

In V-mode, up to 64 key fields may be specified and the total length may not exceed maximum record length.

The following additional key type may be specified as a parameter in the SORT subroutines:

Key	BYTE LENGTH	RANGE
UNSIGNED INTEGER	2	Ø to 65535

For the PACKED DECIMAL key, a negative sign is represented by a hex D in the sign nibble; all other 4-bit combinations in the sign nibble represent a positive sign.

Collating Sequence

ASCII keys may be sorted using the EBCDIC, rather than ASCII collating sequence. The sequence is specified in the spcls parameter (described in SRTF$S and SETU$S.

VSRTLI Subroutines

The following parameters should be changed in SUBSRT, MRG1$S, ASCS$$, SRTF$S, and SETU$S:

numkey   Number of pairs of starting and ending columns (starting and ending bytes if binary). Maximum is 64, default=1.

ntype   Vector containing type of each key. The additional ntype for Rev. 17.6:

Type	
13	UNSIGNED INTEGER

Spcls Parameter

The following parameter should be expanded in SRTF$S, SETU$S, and MRG1$S.

spcls   Five word array containing:

spcls(1) = Space option
  Ø = include blank lines in sort
  1 = delete blank lines
  Default = Ø

spcls(2) = Collating sequence specification
  1 = ASCII collating sequence
  2 = EBCDIC collating sequence
  Default = Ø ASCII collating sequence

spcls(3) = Tag/non-tag option (For MRG1$S, must be Ø)
  1 = Tag sort
  2 = Non-tag sort
  Default = Ø Tag sort

spcls(4-5) must be zero and are reserved for future use.

Subroutines

MRG2$S

MRG2$S returns the next merged record. MRG2$S should not be called for output files.

CALL MRG2$S (rtbuff, length)

rtbuff — Buffer containing next merged record (returned). Should be large enough to hold longest record merged.

length — Length of record (in characters) returned. Once all records have been returned, calls to MRGs$S return a length of Ø.

MRG3$S

MRG3$S closes all units opened by the merge routines. MRG3$S should not be called for output files.

CALL MRG3$S

VMSORT

VMSORT, a V-mode version of MSORTS (in-memory sort library) is now available. The PTABLE argument (common to more than one subroutine) in VMSORT is a two word argument. The R-mode version is still MSORTS and there is no change in the contents of that library.

****

ADD TO PAGE 13-3

Tables 13-2 "Logical Device and Numbers" should be expanded to include the information that FORTRAN unit number 29 is Funit 17, FORTRAN unit 3Ø is Funit 18, ... FORTRAN unit 139 is Funit 127.

****

CHANGE TO PAGE 17-1

Revise the description of O$ADØ7 to read: Multiple blank characters are replaced with '221, followed by a wordcount.

****

CHANGE TO PAGE 19-7

Change the number of words argument in the calling sequence for T$VG to read the limit is 512 words.

****

ADD TO PAGE 19-7

MEGATEK

PRIMOS supports up to four Megatek devices. These devices are ASSIGNED as MGØ - MG3. The user-callable driver is T$MG.

T$MG

```
    CALL T$MG (unit, key, auxdat, buffer, buflen, statv)

unit      unit number (Ø-3)

key       action key
          1 = Read MEGRAPHIC memory (DMX)
          2 = Write MEGRAPHIC memory (DMX)
          3 = Peripheral read
          4 = Peripheral write
          5 = Poll/wait for interrupt
          6 = Reset display
    '10000x = Do physical I/O
              X = 1 OCP
              X = 2 SKS
              X = 3 INA
              X = 4 OTA

auxdat    Megatek memory address
                    or
          if key = 5, auxdat specifies whether waiting,
          polling or timed wait.
              <Ø = waiting
               Ø = polling
              >Ø = timed wait
                    or
          if key = Do physical I/O, auxdat is function.

buffer    Data buffer for DMC

buflen    length of buffer (up to 512Ø words)

statv     3 word status vector
          statv(1)     status flag
          statv(2)     hardware status word
          statv(3)     number of words transferred
```

VECTOR GENERAL

The system supports up to four Vector General devices. These devices are ASSIGNED as GSØ - GS3. The user-callable driver is T$GS.

T$GS

CALL T$GS (unit, key, func, buff, buflen, code)

<u>unit</u>	logical unit number (Ø-3)
<u>key</u>	action key '1Ø = poll for interrupt '11 = wait for interrupt '2Ø = execute OCP '21 = execute SKS '22 = execute INA '23 = execute OTA '3Ø = DMC output '31 = controller busy check
<u>func</u>	Vector General buffer address or Physical I/O function field.
<u>buff</u>	data word or LOC(BUFFER) for DMC
<u>buflen</u>	length of buffer (words)
<u>code</u>	non-standard error code Ø OK (not busy for key = 31) 1 unit not Ø thru 3 2 unassigned unit 3 invalid key 4 bad PIO function (keys '2Ø-'23) 5 unit busy (keys '3Ø-'31) 6 no skip outputting channel address (key '3Ø) 7 no skip preparing for DMC (key '3Ø) 8 no skip outputting buffer address (key '3Ø) 9 no skip on SKS, INA, OTA function keys (keys '2Ø-'23) 1Ø invalid DMC count (buflen) (key '3Ø)

****

ADD TO PAGE 19-24

CHANGES TO T$MT

<u>Instruction to get controller ID</u>

A new command has been added to T$MT to return the controller identification word. The controller ID may be used by software that intends to support all tape drives, but takes advantage of special

features that are available only with a particular controller. For example, the erase command is only available with version 2 and 3 controllers.

CALL T$MT (UNIT,INSTR,BUFF,BUFLEN,STATV,CODE)

INSTR = :140000
BUFF(1) = Controller ID

```
|0      3|4      8|9      16|
-----------------------------
|not used |Revision |Device ID |
-----------------------------
```

Revision = microcode E.C. level
Device ID = bits 9,10 - version of controller
bits 11-16 mag tape controller (:14)

Additional status words

At Rev. 17, the size of the STATV array was extended if the optional CODE argument was present. T$MT has been changed to store the additional hardware status in this array.

STATV Status vector. If this is the last argument, then only the first three words are set. If the CODE argument was given, then the additional words may be set depending on the controller being used.

(1) 1 operation started; 0 = operation done.
(2) hardware status for all controllers
(3) number of words transferred
(4) hardware status for version 1, 2, and 3 controllers.
bits 0,1 = density of tape. (00 = 800 bpi,
10 = 1600bpi, 11 = 6250bpi)
(5,6) hardware status for version 3 controllr.

CODE Standard error code. If this optional argument is used, then the STATV must be a 6 word array. If this argument is not used, then any illegal instructions will result in an error message being printed, and a return to command level.

****

ADD TO PAGE 19-26

Add to subroutine T$MT

Bit 16 of status word 2 does not have any function with tape formatter 2271-901.

ADD TO PAGE 2Ø-2

T$SLCØ

The range of the line parameter is extended:

line   Octal line number Ø-7

****

ADD TO PAGE 21-1

NAMED SEMAPHORES

On timesharing systems where more than one process can be active at the same time, there is often a need to coordinate the execution of multiple processes with one another. Two times that such coordination is required are when two or more processes cooperate to solve a common problem; and when multiple processes must use a common, limited resource.

When multiple processes are working together as a part of a larger system or to solve a common problem, it sometimes happens that one or more of the processes encounter a situation in which they can not do any further work until some event, external to the process, happens. An example of this is a spooler which picks up print requests from a queue. When there are requests in the queue, the spooler services them; however, when the queue becomes empty, it can no longer do useful work and must wait for another process to give it something to do.

There are many resources on a timesharing system that must be shared by all of the running processes. Included in the list are such things as devices that can have only one user at a time (e.g., a paper tape punch), a section of code that performs a single atomic operation, or files that are updated and read simultaneously by several programs.

The semaphore facility provides a means to coordinate multiple processes, providing that the processes involved all use the facility in the same way.

The semaphore facility consists of some blocks of memory, which are called semaphores, and a set of software routines or hardware instructions that perform various operations on these blocks. Note that there is no real connection between a semaphore and the event or resource with which it is associated. The use to which a semaphore is put is determined solely by the application programs that use it. All of the cooperating programs must agree on the meaning (or use) of a semaphore and use it the same way.

The two basic operations that can be performed on a semaphore are called WAIT and NOTIFY, which will be described later. There are other

operations that can be done with semaphores, and they will also be described.

USING SEMAPHORES

The operating system maintains a pool of semaphores which it can assign to user processes. When a processes wishes to use one or more semaphores, it must first ask the operating system to assign them to the process. The process requests access to semaphores via an open semaphore routine. The user can request that multiple semaphores be assigned to it in a single call to this routine. The operating system will return a set of numbers to the process if it decides that the requested semaphores can be assigned to that process. The process will use these numbers in all subsequent calls to semaphore routines to indicate on which semaphore to perform the semaphore operation.

The operating system can tell when different processes wish to use the same set of semaphores by examining the parameters that they include in the call to the open routine.

[see the section on Semaphore Functions: Open Call for more details on how to use the open call].

After a process has opened a set of semaphores, it can do any number of semaphore operations on those semaphores. The possible semaphore operations are described is the section entitled Semaphore Functions.

When a process has finished using the semaphores that were assigned to it, it requests that the operating system CLOSE those semaphores, thus making them inaccessible to the process, unless another OPEN is done. When all processes that were using a semaphore close it, then the space in the operating system taken up by that semaphore is returned to the operating system free pool and may be assigned to other processes at a later time.

When a process logs out, all semaphores that were opened by the process but not closed are closed automatically. If this process was the last user of a semaphore, the space used by the semaphore is returned to the free pool.

DESCRIPTION OF SEMAPHORES

A semaphore consists of two parts: a counter and a queue.

When a process wishes to wait for an event to happen or a resource to become available, it issues a WAIT call for the semaphore associated with that event or resource. The WAIT call will increment the counter for that semaphore and test its value. If the counter is less than or equal to zero, the process is allowed to proceed immediately and is not placed on the semaphores queue. If, however, the count is greater than or equal to one after being incremented, then the process is placed on the wait queue for the semaphore. The process will not run again until

it leaves this queue. Processes are placed on the queue in priority order with higher priority processes being placed closer to the head of the queue. Within a given priority, the processes are queued first in - first out.

When a process wishes to report that an awaited event has occurred, or that a resource has become available for use by other processes, it will call a NOTIFY routine for the semaphore associated with that event or resource. The NOTIFY routine will first test the value of the counter for that semaphore. If the counter is greater than zero (indicating that one or more processes are in the semaphores queue), then the routine will remove one process from the top of the queue, thereby allowing that process to run again. Whether a process was dequeued or not, the routine will then decrement the counter by one.

Normally, a semaphore's counter is preset to some value before the semaphore is used by any process. The value to which it is set depends on the nature of the software that will use the semaphore and on the purpose of the semaphore. Typical initial values are -1 and Ø. A value of -1 allows the first process that waits on the semaphore to proceed immediately without being queued. This effect is desirable if the semaphore is used to coordinate the use of a shared resource. The resource is considered available until a process indicates its intent to use it. A value of zero is appropriate for wait for event situations in which a process must wait until some condition exists or until an event occurs. The process that must wait for an event to happen does a wait operation on the semaphore, and is immediately put on the queue since the counter becomes greater than zero. When another process determines that the awaited event has occurred, it will notify the same semaphore, thus allowing the queued process to run.

When a process opens a semaphore, and that process is the first to open that semaphore, then the open routine will preset the semaphore's counter to a value of zero. If an initial value of -1 is required, then the process should notify the semaphore once after opening it. If the semaphore must be reset to its initial value of zero at a later time, then a call can be made to the drain semaphore routine.

[see Semaphore Functions: Drain Call]

PRIME SEMAPHORES

On Prime computers, a semaphore consists of two (16 bit) words of memory. Any two consecutive words of memory can be used as a semaphore, but these words must be non-pageable. The WAIT and NOTIFY operations are implemented in firmware and are usable by ring Ø software only. So that users can use the semaphore facility, ring three calls have been created that perform the wait and notify operation on a set of semaphores that are reserved by the operating system for use by user programs.

Currently, there are 1Ø24 semaphores available to user processes.

CODING CONSIDERATIONS

Cooperation of Processes

It should be remembered that a semaphore is a method that cooperating processes can use to control their access to resources, or to coordinate their execution. The operating system does not verify that the semaphore is being used correctly since the association between the semaphore and the event or resource is merely a convention adopted by the processes involved.

In order for the semaphore facility to work correctly, all processes that want to wait for an event or a resource must first wait on its associated semaphore before using the resource or assuming that the awaited event has occurred. Since the operating system does not get involved in the internal affairs of the processes (at least as far as semaphore usage is concerned), there is nothing to stop the careless programmer from using a shared resource, for instance, without first waiting on the appropriate semaphore. Such coding practices will most likely cause the entire subsystem of processes to malfunction.

Named vs. Numbered Semaphores

There are two methods by which a process can specify which semaphores it intends to use. Also, there are two sets of semaphores maintained by the operating system. One set is available to any process that wishes to use it, and are identified by number. When a process wishes to use one of these semaphores, it specifies the number of the desired semaphore in the parameter list of the semaphore routines. This set of semaphores is called numbered semaphores. Numbered semaphores are easy to use; however, they have a major drawback: there is nothing to prevent other processes from using the same semaphore for different purposes. Therefore, all users of the system must agree on the usage that each numbered semaphore will have; otherwise, confusion will result.

To eliminate the problems caused by the sharing of numbered semaphores, a second set of user accessible semaphores was created. Semaphores in this set can not be used by a process until they are opened. Opening a semaphore means that the process must call a routine (called SEM$OP) which will assign semaphores from the pool for the process to use. The routine returns a set of numbers which can be used instead of numbered semaphore numbers in all other semaphore rotine calls. Only valid numbered semaphore numbers and semaphore numbers that have been assigned to a process by SEM$OP can be used in subroutine calls that manipulate semaphores. An attempt to use any other numbers will result in an error return from the routine.

So that multiple processes can use the same semaphores for coordinating their activities, the process must supply a unique name to the SEM$OP routine which will be used to uniquely identify the semaphore. The name supplied must be the name of a file in the UFD attached by the process. All calls to SEM$OP that specify the same file will result in

the same semaphore numbers being returned.

Timers and Timeouts

When a process waits on a semaphore, it anticipates that it will be notified within a reasonable amount of time. If for some reason, the process that is going to notify the semaphore fails to do so, all processes waiting on that semaphore will continue to wait, possibly for a very long time.

To guard against processes waiting forever, a timer mechanism is used.

Named Semaphore Timers: To prevent a process from waiting forever on a named semaphore, a special wait routine exists (called SEM$TW) which takes a semaphore number and a time value as parameters. The process will wait on the specified semaphore until the semaphore is notified or until the specified amount of real time has passed. The routine returns a value to the process that indicates why the process was allowed to continue. A value of zero (Ø) means that the semaphore was removed from the wait queue because of notify by another process. A value of one (1) means that the process was allowed to continue because the the specifed time had elapsed without a notify on that semaphore. It is also possible for a value of two (2) to be returned. This return value indicates that the process was stopped by someone typing the break key, or control-P, at the terminal controlling the process, and then typing START. This sequence of things causes the operating system to abort the process, thus removing it from the semaphore on which it was waiting; followed by a restart of the process at the instruction following the wait call. If a special value were not returned to the process in this case, the process may think that the semaphore on which it was waiting was notified by another process -- which is not the case. The special return value allows the process to detect this situation and to rewait on the semaphore, if that is appropriate.

Numbered Semaphore Timers: The timer facility for numbered semaphores allows a semaphore to be automatically notified after a certain amount of time has passed. A user process tells the operating system, via a subroutine call, that a timer is to be associated with a numbered semaphore. The process also specifies the amount of time that should pass before the operating system notifies the semaphore. When this amount of time has passed, the operating system notifies the semaphore.

Much care is needed when coding programs that use semaphores with this kind of timer; if another method is not used besides the semaphore to indicate that the awaited event has actually occurred, then a notify caused by a timer can not be distinguished from a notify caused by a process. The processes using the semaphore should, therefore, be coded so that they can verify that a notify by another process has occurred before using the resource protected by the semaphore. The action that is taken when a timer notifies the semaphore is subsystem dependent and should be agreed upon by all of the processes using the timed

semaphore.

PITFALLS AND HOW TO AVOID THEM

External Notifies: When a semaphore is notified for some reason other than an explicit call to the notify routine, that notify is called external; that is, it originated from a source external to the processes that are using the semaphore. Some of the reasons that a external notify may occur are listed here.

Expiration of a Timer: When a timer is set for a numbered semaphore, and that timer expires, the operating system will notify the semaphore. This semaphore will look like an external notify to the processes that use the semaphore; the fact that the notify is external can be detected if the processes are coded properly (see coding suggestion, below).

The notify caused by a time out can be useful in cases when the procses that is supposed to notify the semaphore is prone to being aborted. The operating system initiated notify will prevent processes from waiting forever.

Use of timers with named semaphores causes a code to be returned to the process that indicates when a timeout has occurred.

Malfunctioning Process

Processes that are supposed to be using a semaphore, like all other programs, sometimes do not behave properly. Malfunctioning programs can do extra notify calls, and cause what appears to be external notifies. Also, processes that are not supposed to be using a numbered semaphore may decide to use it anyway. Unless the semaphore can be protected from such interference, then what appears to be an external notify will result.

Process Quit

The semaphores that a user process can access on a Prime system are called quittable semaphores. This means that a process that is waiting on a semaphore can be stopped by typing <BREAK> or <CONTROL-P> at the terminal controlling the process. When a process is stopped by this means, and then continued (by typing the PRIMOS START command), the operating system will remove the process from the semphore wait queue so that it can be returned to system command level. If the program is them continued, it will not reexecute the wait operation. The process will continue to run and will believe that it was notified from the semaphore by another user process.

Since semaphores can be notifed by breaks and timeouts as well as by

explicit calls to SEM$NF, and since this could cause malfunctions in a sub-system, it is always best to code in such a way that this situation can be detected. This means that a process should not rely solely on the semaphore to indicate that a resource is really available or that an event has actually occurred. A good practice is to have one additional method, besides the semaphore, to indicate what the current state of the resource or event is.

One such method is to have a word in shared memory (accessible by all cooperating processes) which is set to indicate that the event has really occurred or that a resource is free. Before a process notifies a semaphore, it sets this word to an agreed upon value. When the process is allowed to proceed from a semaphore wait, it should check the value contained in that word. If the word contains the agreed upon value, it will know that the semaphore was notified by a cooperating process, and not by the operating system. In this case, the process will clear the word, do its processing, and reset the word to the agreed upon value just before notifying the semaphore. If a process proceeds from a wait call and the word is not set to the agreed upon value, it can assume that the operating system notified the semaphore and can reissue the wait call.

Infinite Waits: It is possible to create a situation in which one or more processes are waiting on a semaphore, and there are no processes running that will ever notify that semaphore. Methods of creating this situation include:

Multiple Waits

If a process issues a wait call, and is NOT queued, and then continues to reissue the wait call without intervening notifies, that process will eventually cause the semaphore count to become greater than zero and the process will wait. This of course assumes that there is not another processes somewhere doing multiple notifies!

In the case of a resource protecting semaphore, if all other processes obey the rules, they will wait on this semaphore before they notify it. They will therefore queue up behind the multiple waiter process. Eventually, all the processes of the sub-system will become queued on the semaphore queue, and no process will remain to notify the semaphore.

Aborted Notifiers

Another way of causing infinite waits is to abort a process that would, under normal circumstances, notify a semaphore. If this is the only process that will do notifies on the semaphore, then all other processes that wait on it will wait forever.

Infinite waits can be avoided by associating a timer with the semaphore. This will guarantee that one or more processes will

eventually be removed from the wait queue. Extra coding must be done in the processes, however, so that a time out can be identified as such, and so that appropriate action can be taken. This code should determine whether the process that should have notified the semaphore is still running or not. If it is running, the notify is considered external and the process reissues the wait call. If the potential notifiers have all been aborted, appropriate recovery action should be initiated.

Deadly embrace: When multiple semaphores are being used, a situation called deadly embrace can occur. Ths happens when two processes each gain rights to use a resource by waiting on the appropriate semaphore for that resource, and then each attempt to acquire the resource that is being used by the other process. Clearly, neither process will ever notify the semaphore for the resource they hold (they are waiting to get access to a second resource), and no other process will ever notify the semaphores (since each resource is held already by one of the two processes). Therefore, both processes will wait forever.

This situation can not be detected, nor can it be prevented by the semaphore facility. It can be prevented, however, by the processes using the semaphores if the following procedure is used.

Each semaphore that a system of processes will use is assigned a different number; this number will be called the semaphore's level number. Processes can only issue a wait call for a semaphore whose level number is greater than the level number of any semaphores it has waited on but has not yet notified. For example, if the level numbers for three semaphores are 1,2, and 3, and a process has waited on the second semaphore (level 2) but has not yet notified it, then the process can legally issue a wait for the third semaphore (level 3) but not for the first since level 1 is numerically less than level 2.

This technique, if strictly followed, makes deadly embrace situations impossible. It is sometimes practical for processes to call a routine which checks for level number violations and then issues the wait call if it is ok to do so. If all processes use this routine, instead of the wait routine, then deadly embrace is prevented.

SEMAPHORE FUNCTIONS (PRIME MACHINES)

The following semaphore operations are available to user processes. Note that parameters that are not underlined in the calls are passed to the semaphore routine; parameters that are underlined are returned to the process by the routines.

Although data types for parameters are specified in PL/I terms, the routines are also callable from FORTRAN programs.

Open Semaphores

CALL SEM$OP (fname, namlen, nbr, ids, code)

Fname (char(32)) is a file name, discussed below.

Namlen (fixed bin) is the number of characters in fname.

Nbr (fixed bin) is a number that specifies how many semaphores are to be opened by this call.

Ids ((nbr)fixed bin) is an array of semahore numbers; one number is returned for each semaphore that was successfully opened.

Code (fixed bin) is a success/failure code. A value of Ø indicates success; E$BPAR indicates that an invalid value was supplied for snbr; E$IREM means that a file that is on a remote disk was specified in the fname parameter -- remote files can not be used as parameters to this call; E$FUIU means that either the user has all available file units opened, or that there are no available named semaphores. It is also possible that code will be set to any error code that can be returned by the SRCH$$ routine.

On Prime systems running Rev. 19, or later, of PRIMOS, it is possible for a number of processes to have access to a set of semaphores while other processes are denied access to the same semaphores. These semaphores are called protected or named semaphores.

To access a named semaphore, a call must be made to SEM$OP, which grants or denies access to the semaphore. The process supplies a file name to the call. If the specified file can be accessed for read access, subject to file system and ACL protections, then the user is given access to the requested semaphores. Multiple semaphores can be opened in a single call by supplying the number of semaphores needed in the nbr parameter.

If access is granted to the semaphores, then the call will return an array of semaphore numbers in the ids parameter. One number will be returned for each semaphore requested in nbr, assuming enough semaphores exist in the system pool. A semaphore number of zero will be returned if a semaphore could not be assigned. In addition, code will be non-zero if one or more semaphore numbers could not be assigned. The values returned in ids should be examined to determine which semaphores were opened (non-zero value returned), and which were not (zero value returned).

The semaphore numbers returned should be used in all other semaphore calls as the semaphore number parameter. SEM$OP is the only call that takes a file name and returns semaphore numbers; the rest of the calls accept only a semaphore number.

If different processes call SEM$OP and specify the same file, the same semaphore numbers will be returned to each process. This allows

multiple processes of a subsystem to reference common semaphores.

If a call to the open routine specifies the same file as a previous call to open, and a larger number of semaphores is requested, then new semaphores are acquired from the system pool to make up the difference between the number currently open (with that file name) and the number requested in the call. Other processes can not use these newly assigned semaphores unless they explicitly open them via a call to the open routine.

When the first process opens a named semaphore, the operaing system will set the value of the semaphore counter to zero (Ø). Subsequent opens of the semaphore do not alter the value of the counter. If a process opens the same semaphores more than once, then the same semaphore numbers will be returned for each call. No matter how many times a process opens a semaphore, it need only close that semaphore once. This removes the burden of counting open and close calls to being sure that equal numbers of calls are done.

Named semaphores can only be opened for files that reside on a local computer system. Attempts to open named semaphores with file names that are on remote disks will result in failure; no semaphore numbers will be assigned and code will be set to E$IREM.

If a file that was used in a call to SEM$OP is deleted or renamed while the semaphores assigned by such a call are still open, or if the disk on which that file resides is shut down, then the open semaphores will continue to be accessable to the processes that already have them open. New processes will not be given access to those semaphores, even if the disk is added again, or if a file is created with the same name as the one that was renamed or deleted. Processes that have the semaphores open can continue to use them until they are closed via a call to SEM$CL.

If a process logs out before all named semaphores have been closed, then those that are still open will be automatically closed by the operating system.

Notify/Wait

CALL SEM$NF (snbr, code)

CALL SEM$WT (snbr, code)

snbr (fixed bin) is a semaphore number; it can be either a number in the allowable range for numbered semaphores (1 - 64), or it can be a number assigned to a named semaphore by the SEM$OP routine.

code (fixed bin) is a success/failure code returned by the routine. A value of Ø indicates success; E$BPAR indicates that an invalid value was supplied for snbr.

As explained in an earlier section, the notify and wait operations are the basic functions that can be performed on a semaphore. NOTIFY decrements the semaphore's counter and will release the first process from the wait queue, if there are any processes waiting.

WAIT increments the semaphore's counter and places the process on the semaphore's queue if the counter becomes greater than zero. Processes are queued first in - first out within process priority; higher priority processes are queue before those with lower priority.

Test

sval = SEM$TS (snbr, code)

sval (fixed bin) is the current value of the specified semaphor's counter word.

snbr (fixed bin) is a semaphore number; it can be either a number in the allowable range for numbered semaphores (1 - 64), or it can be a number assigned to a named semaphore by the SEM$OP routine.

code (fixed bin) is a success/failure code returned by the routine. A value of Ø indicates success; E$BPAR indicates that an invalid value was supplied for snbr.

This operation returns the current value of the counter, for semaphore numbered snbr in the variable sval.

The values of code are the same as for WAIT and NOTIFY.

Drain

CALL SEM$DR (snbr, code)

snbr (fixed bin) is a semaphore number; it can be either a number in the allowable range for numbered semaphores (1 - 64), or it can be a number assigned to a named semaphore by the SEM$OP routine.

code (fixed bin) is a success/failure code returned by the routine. A value of Ø indicates success; E$BPAR indicates that an invalid value was supplied for snbr.

This operation resets the specified semaphore counter to zero. If, at the time of the drain call, the semaphore's counter is less than or equal to zero, the counter is set to zero. If, however, the counter is greater than zero, then notifies are done on the semaphore until the counter reaches zero. This causes all processes that were waiting on the semaphore to be removed from the wait queue of the semaphore.

It is possible for processes to be placed on the wait queue while this

call is executing; these added processes may not be removed when the drain call returns to its caller.

Set Timer (Numbered Semaphores)

This operation causes the operating system to notify the specified semaphore on a periodic basis.

CALL SEM$TN (snbr, intl, int2, code)

snbr (fixed bin) is a semaphore number; it must be a number in the allowable range for numbered semaphores (1 - 64).

intl is the amount of clock time (in milliseconds) that will pass before the system notifies the semaphore the first time.

int2 is the amount of clock time that will pass before the semaphore is notified the second and subsequent times. If int2 is zero, then the semaphore will only be notified once - after intl milliseconds. Specifying both intl and int2 as zero will remove a previously timer request from the semaphore. This is necessary when a previous SEM$TN call was made with intl and int2 both non-zero.

If a call is made to SEM$TN which specifies a semaphore that already has an active timer request, then the values of intl and int2 specified in the latter call will overwrite the values stored in the active timer. Note: it is possible to delay a timeout initiated notify of a semaphore indefinitely by making repeated calls to SEM$TN.

code (fixed bin) is a success/failure code returned by the routine. The values of code are the same as those returned by WAIT and NOTIFY.

The operating system maintains a limited number of timers for numbered semaphores. Currently, there are a total of fifteen (15) such timers per system.

Timed Wait (Named Semaphores Only)

CALL SEM$TW (snbr, intl, code)

snbr (fixed bin) is a semaphore number; it must be a number assigned to a named semaphore by the SEM$OP routine.

intl (fixed bin) is a time interval expressed in tenths of a second of clock time.

code (fixed bin) is a success/failure code returned by the routine.

This routine allows a process to wait on the specified semaphore until it is notified off of the wait queue, or until a specified amount of

real time has elapsed, whichever comes first. code is set to a value which indicates why the process was allowed to continue. A value of zero (Ø) indicates that the process was notified by a call to SEM$NF. A value of one (1) indicates that the specified amount of time has elapsed and the process has not yet been notified off of the wait queue. A value of two (2) indicates that the process was aborted from the controlling terminal by control-P or <BREAK> being typed and that the process was then continued with the PRIMOS START command.

Close Named Semaphore

CALL SEM$CL (snbr, code)

snbr (fixed bin) is a semaphore number; it must be a number assigned to a named semaphore by the SEM$OP routine.

code (fixed bin) is a success/failure code returned by the routine.

When a process no longer needs a named semaphore, it can tell the operating system that it is done with it by calling SEM$CL, to CLOSE the semaphore. After this call, the specified semaphore number can not be used again by the process, unless that same number is reassigned by another call to SEM$OP.

When a process logs out, all semaphores that were opened by that process, but not explicitly closed, are automatically closed by the operating system.

It is only neccessary to close a named semaphore once in a process regardless of the number of times that it was opened.

If a file that was used in a call to SEM$OP is deleted or renamed while the semaphores assigned by such a call are still open, or if the disk on which that file resides is shut down, then the open semaphores will continue to be accessable to the processes that already have them open. New processes will not be given access to those semaphores, even if the disk is added again, or if a file is created with the same name as the one that was renamed or deleted. Processes that have the semaphores open can use them until they are closed via a call to SEM$CL.

LOCKS

For a detailed description of locks, please consult a text on operating system coding techniques. A brief discussion follows.

Locks, like semaphores, are a method which programs or processes can use to coordinate their usage of some resource. Before a process attempts to use a resource that is protected by a lock, it calls a routine that grants or delies permission to use the resource or causes the process to wait until the resource becomes free. When the process has been given permission to use the resource, it is said to hold the

lock on that resource. When the process is through using the resource, it calls another routine to indicate that it is done. This operation is called giving up the lock, or releasing the lock on that resource.

Various types of locks exist, some of which will be discussed in this section.

Some types of locks behave very much like semahores and, in fact, many types of locks can be coded with the use of semaphores. Semaphores, unlike locks, allow a small, well defined set of operations to be performed while the uses and types of locks that can be coded vary greatly.

Mutual Exclusion

Mutual exclusion locks are used when only one process, or possible several, is allowed to use a resource at any given time. When a process requests ownership of a lock for the resource, it is given the lock if no other process currently holds it. If the lock is held by another process, all others must wait until the one holding the lock gives it up.

This type of lock can be implemented directly with the use of semaphores. Requesting the lock is equivalent to a wait operation on a semaphore; giving up the lock is equivalent to notifying that semaphore.

Since external notifies may occur, it is a good practice to expect them and to code in such a way that they can be detected and ignored.

N1 Locks

N1 locks are used to protect objects that can be both read and modified simultaneously, such as files and data bases. This type of lock allows any number of users to read the object, or one process to modify the object. When a process requests permission to read the object, such permission is granted immediately, as long as there is not currently a process modifying it. Requests to gain access to the object for modification are granted only if there are no other readers or writers of the object. If another process is using the protected object, the writer is placed on a queue and must wait until all current users of the resource indicate that they are done. If a writer is waiting to use the resource, then no other requests for use of the object are granted until that process has used the object. This prevents readers from gaining access to the object and causing the write request to be delayed indefinitely.

When a writer is given access to the object, all other requests for access are queued. When the writer finishes, the other requests are processed.

Use of an N1 type lock on a file eliminates data lossage that can

sometimes occur when multiple processes are allowed to update the same file simultaneously.

Producer-Consumer

In many computer systems, certain processes create work which must be processed, such as device drivers that read data from a device which must be routed to the correct place, or print programs that place data files into spool queues to be printed. These work producing processes are called producers.

Other processes in a system process the work created by the producers. These processes are called consumers. Examples of consumers include a user process that manipulates data coming into the system from a peripheral device, or a spooler that prints files in response to user's print requests.

The coordination required between producer processes and their corresponding consumer processes can be achieved with the use of producer-consumer locks.

Producers call a routine that indicates that there is work to process. The routine keeps track of the number of producerers that have called it; each call indicates that another unit of work is available. Consumers, on the other hand, call a routine that checks to see if there is any . work to do. If there is no work, the routine causes the consumer process to wait until there is work, i.e., a producer calls the "I have work to do" routine. If there is work to do, the consumer process is allowed to continue, and the counter of units of work left to do is decremented.

This lock can be coded directly with semaphores. A semaphore, with its counter initialized to zero, serves as the locking mechanism. Producers notify the semaphore, causing it to become negative; consumers wait on the semaphore, causing it to rise toward zero. If there is no work to do (semaphore counter equal to Ø) then a consumer will be queued, when it waits on the semaphore, until work becomes available.

Note that there can be any number of producers or consumers. If multiple consumers wait for work, and there is none to do, then the semaphore counter will contain a value equal to the number of queued consumer processes. A notify by a producer will allow one of the consumers to proceed.

Since semaphores are subject to external notifies, it is advisable that a counter, other that the counter that is a part of the semaphore, be maintained to indicate how much work is available for consumer processes. Producers will increment this counter; consumers will take work from the work queue and decrement this counter. If a consumer is notified off the semaphore and the counter does not match the semaphore counter, then it can assume that an external notify has occurred.

Record Locks

When many processes must update a file, and speed is important, it is not practical to use a lock which protects the entire file since any update request would lock all other processes out of using the file. Considerable overlap in processing can usually be achieved if just the portion of the file that is being updated by a process is locked. Usual units to lock are the record or page being updated.

If the file is at all large, then it becomes impractical or impossible to have an individual lock for each record or page to be protected. One way of overcoming this difficulty is to assign locks from a pool on a temporary basis. When a process wishes to update a record, for example, it requests a lock by passing the record number in question to the lock routine. If there is currently no one holding a lock on that record (the lock routine scans its list of locks being held by other processes), then a lock is assigned from a free pool and the record number supplied is remembered. If a lock is requested for a record that is currently locked by another process, then the second and subsequent requesters of the lock are forced to wait. When the last holder of a lock gives up the lock, and there are no other processes waiting to use the record protected by that lock, then the lock itself is returned to the pool of free locks. It can then be used for other record locks.

In general, the pool of locks needs to be as large as the expected maximum number of records that can be locked at any given time. It is the get a lock routines responsibility to manage the lock pool and to deal with the problems that arise when there are no more free locks in the pool. One method of dealing with this situation is to use a no free locks semaphore. If there are no free locks in the pool, the process requesting the lock is forced to wait on this semaphore. The lock routine notifies this semaphore when a lock becomes available.

Notice that record locks are really mutual exclusion locks; however, the object that is being protected by any given lock changes with time. The lock routine must include a small data base that is used to remember what is being protected by each lock.

****

ADD TO PAGE 23-1

REENTER$

This condition is raised by the PRIMOS REENTER (REN) command and re-enters a subsystem that has been temporarily suspended due to another condition (such as a QUIT signal).

If the interrupted operation can be aborted, the subsystem's on-unit should perform a nonlocal goto back into the subsystem at the appropriate point.

If the QUIT occurred during an operation that must be completed, the on-unit should set the info.start_sw to '1'b, record the QUIT request within the subsystem and return. The REN command will then execute a START command which will restart the subsystem at the point of interrupt. When the operation is complete, the subsystem should then honor the recorded QUIT request.

The default on-unit returns without setting the info.start_sw. The REN command will then print a diagnostic and return since it assumes there was no subsystem on the stack able to accept re-entry. Information structure:

```
dcl 1 info based

2 start_sw bit(1) aligned;
```

FIXEDOVERFLOW

This condition is detected by hardware and is raised when a fixed-point decimal or binary result is too large to fit into the hardware register or decimal field. The default on-unit prints a message and signals the ERROR condition. User on-units may not return to the point of interruption.

OVERFLOW

This condition is raised when the result of a floating-point binary calculation is too large for representation. It may occur within a register or as a store exception. The default on-unit prints a message and signals the ERROR condition. User on-units may not return to the point of interrupt. However, if the default on-unit is invoked, and if the user types START, the register or memory location affected will be set to the largest possible single-precision floating-point number, and calculation will continue.

UNDEFINEDFILE(file)

This condition is raised when an OPEN statement cannot associate an input file with an existing PRIMOS file or device. The default on-unit prints a message and signals the ERROR condition.

UNDERFLOW

This condition is signalled when the result of the floating-point binary or decimal calculation is too small for representation. The default on-unit sets the floating-point accumulator to o.oeo. If the underflow occurred as a store exception, the affected portion of memory is also set to o.oeo. The default on-unit returns and the calculation proceeds, using the o.oeo value.

ZERODIVIDE

This condition is signalled when a division by zero (floating-point or fixed-point) occurs. The default on-unit prints a message and signals the ERROR condition. If the condition is the result of a floating-point operation, the user may type START following the printing of the message. The default on-unit will then set the register involved to the largest possible single-precision, floating-point value and proceed with the calculation.

****

CHANGE TO PAGE 23-2

FORTRAN CONSIDERATIONS -

The second restriction on nonlocal gotos shoudl read (change is underlined)

- Nonlocal gotos will work only to a previous stack level since the target statement label belongs to the called of the subroutine performing the nonlocal goto.

ADD TO PAGE 23-7

MKON$P

The direct call MKON$P creates an on-unit.

DCL MKON$P ENTRY (char(*), fixed bin, entry);

CALL MKON$P (condname, namelen, handler;

condname	is the name of the condition for which an on-unit is desired. The name should not contaiun any blanks. (Input)
namelen	is the length of condname, in characters. (Input)
handler	is the internal or external entry (subroutine) value which is to be invoked as the on-unit. If the value is an internal procedure, it must be immediately contained in the block calling MKON$P. (Input)

This call functions as MKONU$ and MKON$F: an on-unit for the specified named condition is created for the calling block. If the block already

has an on-unit for that condition, the on-unit is redefined.

This routine will not work properly if called from a FTN program: use MKON$F for this purpose. Conversely, MKON$F will not work in a F77 program: MKON$P should be used.

****

ADD TO PAGE 23-8

MKON$F cannot be called from FORTRAN 77.

****

CHANGE TO PAGE 23-21

Condition Mechanism example Ret PB and Ret SB ptr should both have options short as part of their declaration for compatability. Otherwise all offsets within the structure are shifted.

****

ADD TO PAGE APPG-1

Add the following error codes:

E$BLUE	100	Bad LUTBL entry
E$NDFD	101	No driver for device
E$WFT	102	Wrong file type
E$FDMM	103	Format/data mismatch
E$FER	104	Bad formate
E$BDV	105	Bad dope vector
E$BFOV	106	F$IOBF overflow